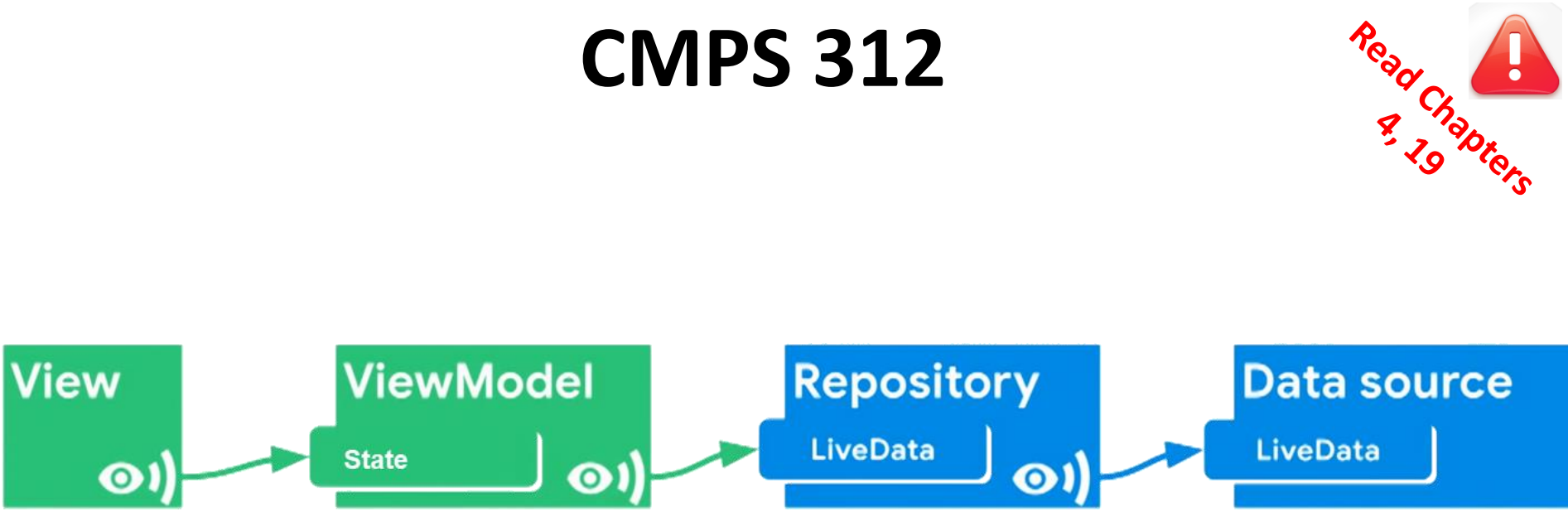
CMPS 312



Model-View-ViewModel (MVVM) Architecture

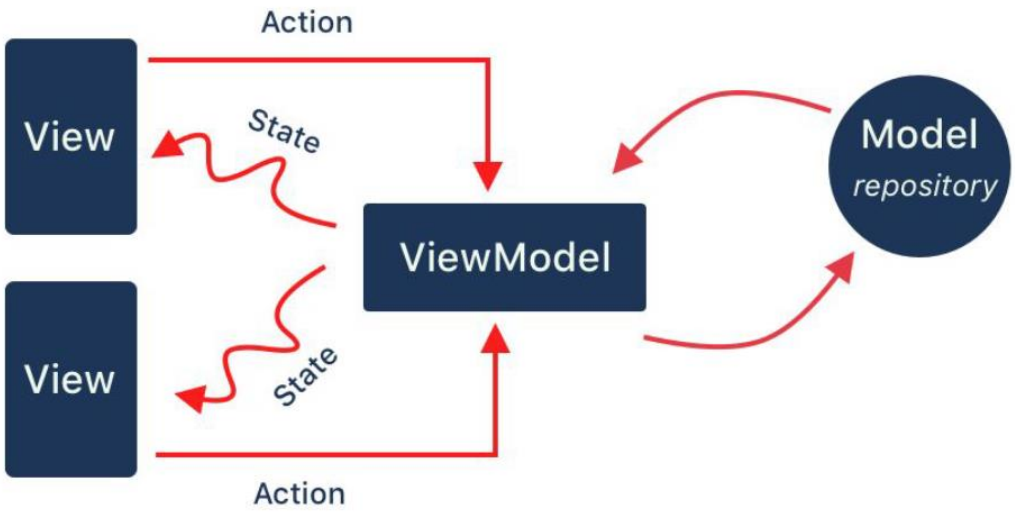
Dr. Abdelkarim Erradi

CSE@QU

Outline

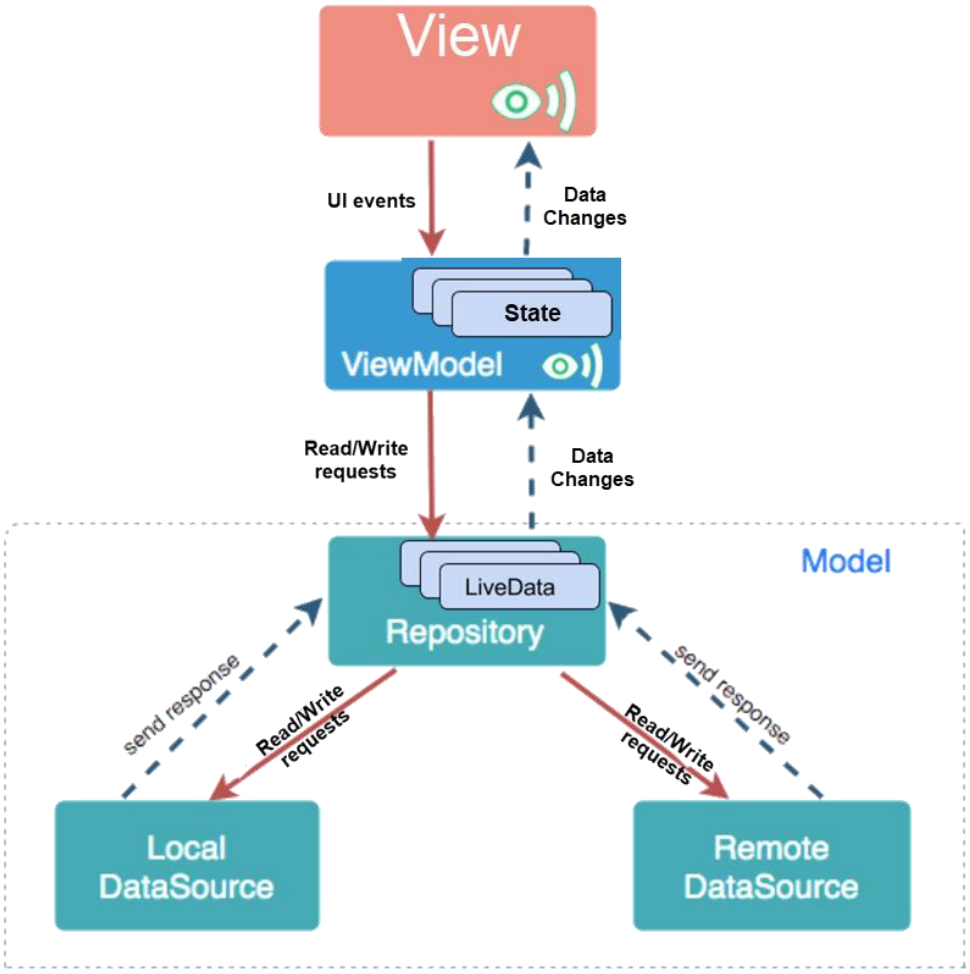
1. Model-View-ViewModel (MVVM)
2. ViewModel
3. State variables
4. LiveData
5. Flow

MVVM Architecture



Back

Model-View-ViewModel (MVVM) Architecture



View = UI to display state & collect user input

- It **observes** state changes from the ViewModel to update the UI accordingly

- Calls the ViewModel to handle events such as button clicks, form input, etc.

ViewModel

- Manages **state** (i.e, data needed by the UI)
  - Interacts with the Model to read/write data based on user input
  - Expose the state as **Observables** that the UI can subscribe-to to get data changes
- Implements UI logic / computation (e.g., data validation)

Model - handles data operations

- Model has **entities** that represent app data
- **Repositories** read/write data from either a Local Database (using **Room** library) or a Remote Web API (using **Retrofit** library)
- Implements data-related logic / computation

MVVM Key Principles

- Separation of concerns:
  - View, ViewModel, and Model are **separate components** with distinct roles
- Loose coupling:
  - ViewModel **has no direct reference to the View**
  - View never accesses the model directly
  - Model unaware of the view
- Observer pattern:
  - View observes the ViewModel (to get data changes)
  - ViewModel observes the Model (to get data changes)

- Inversion of Control:
  - Uses Dependency Injection instead of direct instantiation of objects

*e.g., val scoreViewModel = **viewModel**<ScoreViewModel>()*

Advantages of MVVM

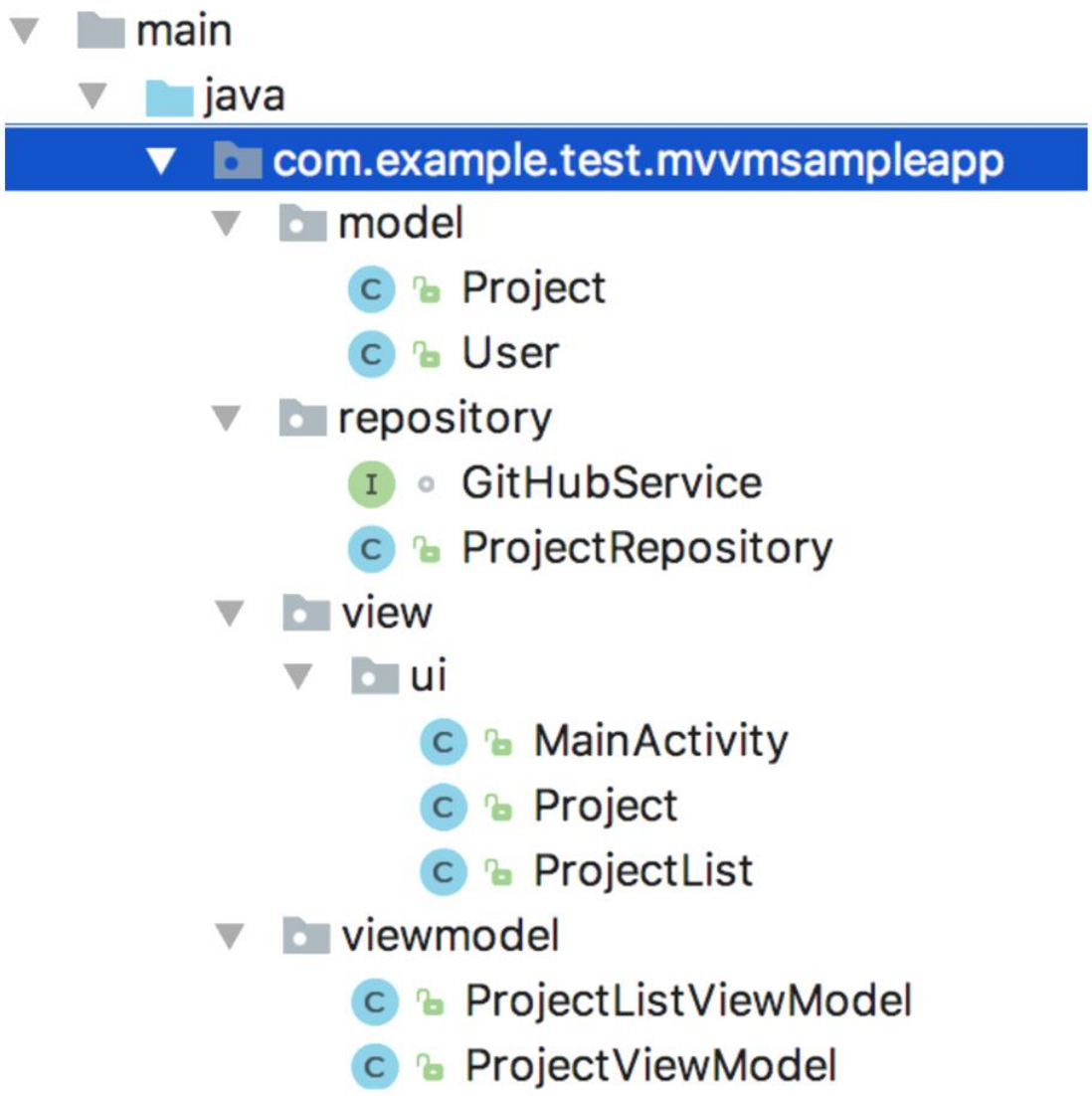
- ***Separation of concerns* = separate UI from app logic**
  - App logic is not intermixed with the UI. Consequently, code is cleaner, flexible and easier to understand and change
  - Allow changing a component without significantly disturbing the others (e.g., View can be completely changed without touching the model)
  - Easier **testing** of the App components

MVVM => Easily maintainable and testable app

Android Architecture Components

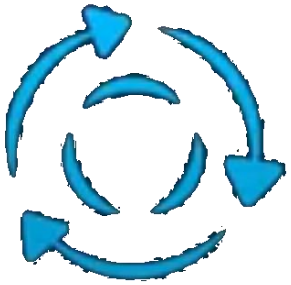
- Android architecture components are a collection of libraries to ease developing MVVM-based Apps
- Part of Android Jetpack  They include:
  - ViewModel stores UI-related data that isn't destroyed on screen rotation
  - LiveData data holder that notifies the ViewModel when the model data changes
  - Room to read / write data to local SQLite database

Recommended Project Structure



You may organize the view by feature

ViewModel

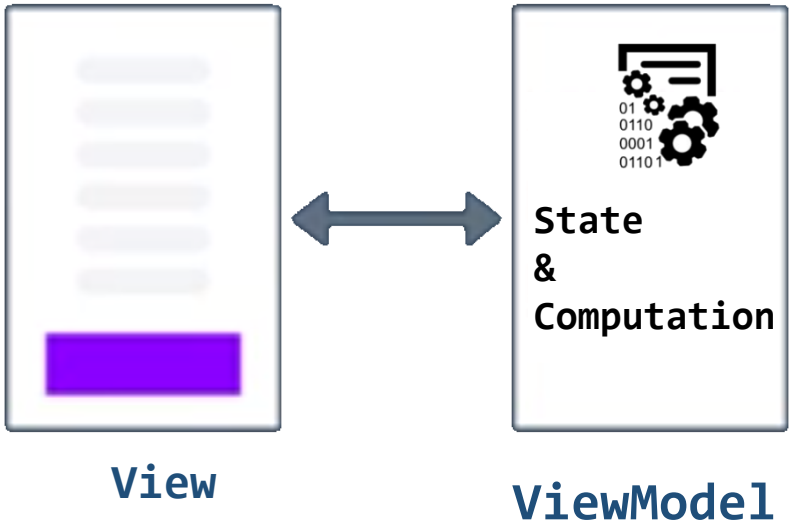


Lifecycle Aware

Survives Config Changes

ViewModel

- ViewModel is used to **store and manage state** (i.e., data needed by the UI)
  - in a lifecycle conscious way
  - allows **state** to survive device configuration changes such as *screen rotations* or *changing the device's language*

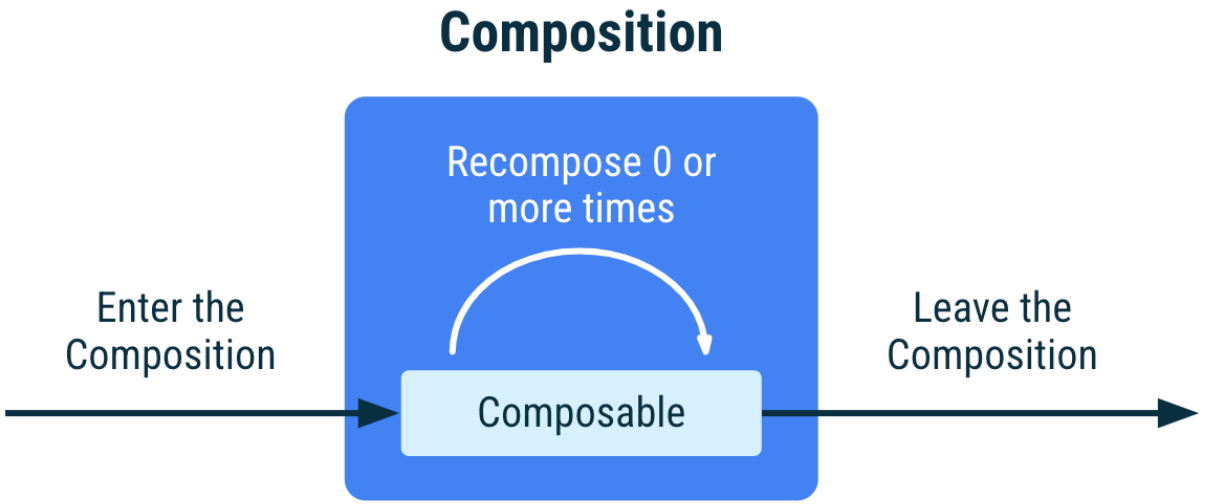


- If the system destroys or re-creates a UI component (e.g., when the screen rotates), any state stored in the View is lost
  - State is NOT retained across configuration changes (landscape/portrait)

Use **ViewModel**:

- Manages state
- Read/write data from a Repository

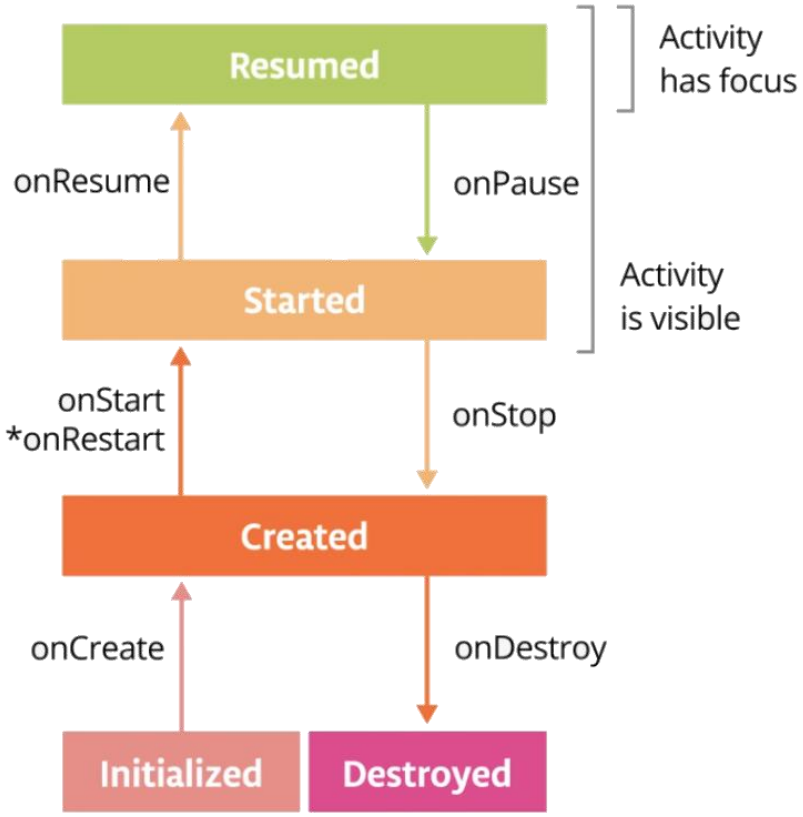
Composable Lifecycle



- When the screen is displayed for the first time, the Composable **enters** the Composition, gets **recomposed** 0 or more times, and **leaves** the Composition
- Recomposition is triggered by a change to a State<T> object. Compose tracks these and runs all composables in the Composition that read that particular State<T>

https://developer.android.com/jetpack/compose/lifecycle

Activity Lifecycle

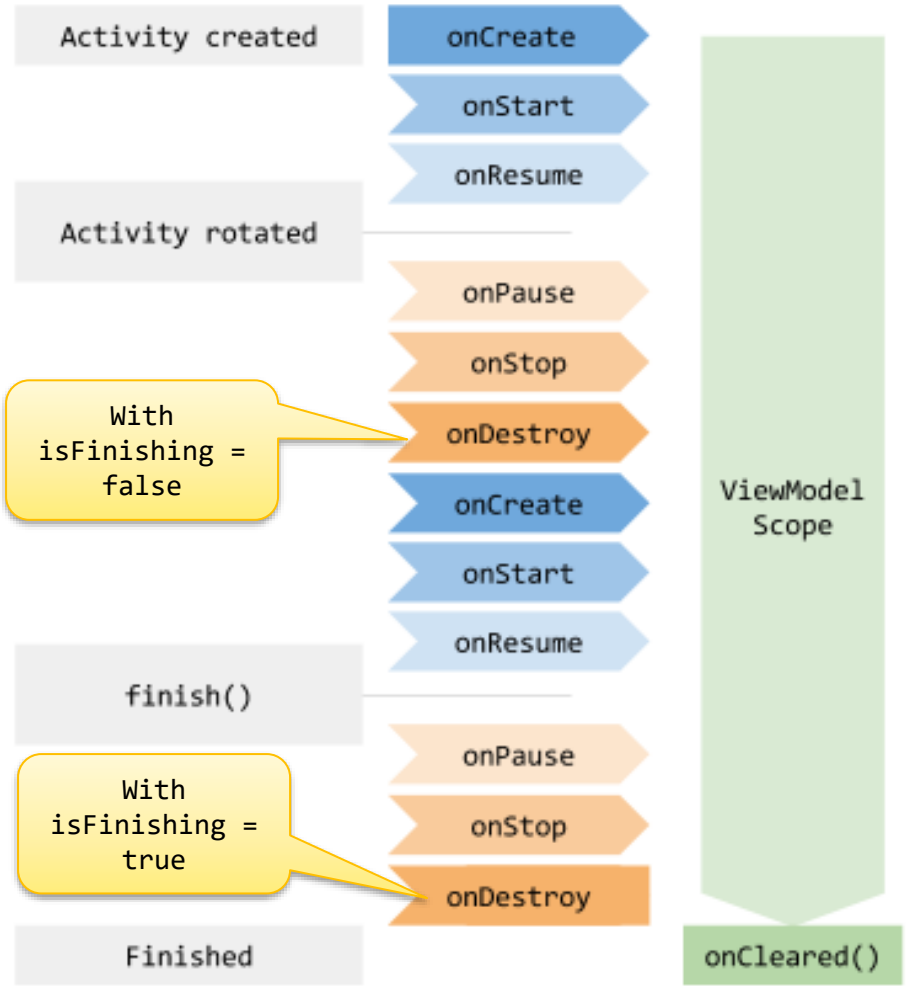


An activity has essentially **four states**:

- ***Resumed*** if the activity in the foreground of the screen (has focus)
- ***Started*** if the activity has lost focus but is still visible (e.g., beneath a dialog box).
  - When the user returns to the activity, it is **resumed**
- **Created** if the activity is completely obscured by another activity.
  - When the user navigates to the activity, it must be **restarted** and restored to its previous state.
- **Destroyed** when the user closes the app or if the activity is killed (when memory is needed or due to `finish()` being called on the activity)

ViewModel Lifecycle

- ViewModel object can be scoped to the main activity
- However, it has a **longer lifespan** compared to the associated Activity which may undergo a rotation and get recreated
- It remains in memory until the activity is `completely destroyed`
  - When the activity is recreated (after a screen rotation) the associated ViewModel remains alive



ViewModel Example

```
class ScoreViewModel : ViewModel() {
    // Private mutable state variables
    private var _team1Score = mutableStateOf(0)
    // Public State variables
    val team1Score : State<Int> = _team1Score
    fun onIncrementTeam1Score() { _team1Score.value++  }
```

```
@Composable
fun ScoreScreen() {
    // Get an instance of the ScoreViewModel
     val scoreViewModel = viewModel<ScoreViewModel>()
    Text(text = scoreViewModel.team1Score.value)
    Button(onClick = { scoreViewModel.onIncrementTeam1Score() }) {
       Text(text = "+1")
    }
    ...
}
```

Add this dependency to build.gradle:

```
implementation "androidx.lifecycle:lifecycle-viewmodel-compose:2.4.0-beta01"
```

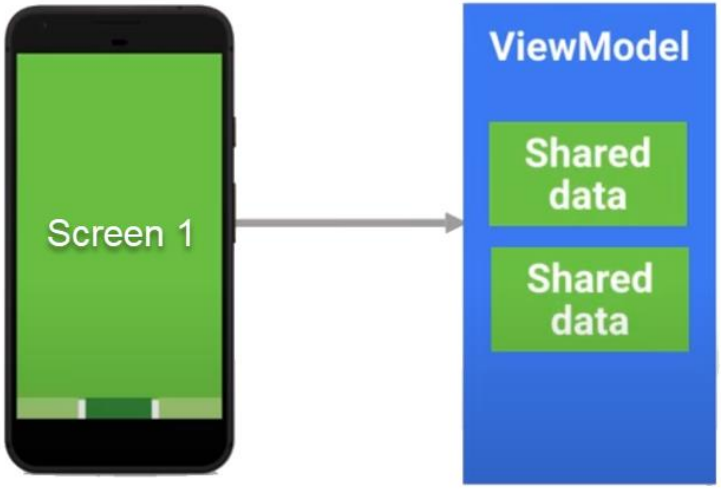
Using ViewModel with Compose Navigation

- By default, Jetpack Compose Navigation associate the viewModel to each destination
  - The viewModel get destroyed when navigating to another destination
- To create a **shared** viewModel that can be used by multiple screens make the viewModel **scoped** to the activity (by assigning the activity as the viewModel Store Owner)

```
/* To allow any screen to get an instance of the shared viewModel
   make the viewModel scoped to the activity (by assigning the activity as
   the viewModel Store Owner)
   This ensures that the same viewModel instance is used for all screens */
val scoreViewModel = viewModel<ScoreViewModel>
        (viewModelStoreOwner = LocalContext.current as ComponentActivity)
val team1Score = scoreViewModel.team1Score
```

Shared data between Screens using ViewModel

- Screens can **share** data using a shared **View Model** class that extends `ViewModel()`




```
@Composable
fun ProfileScreen(userId: Int) {
 /* Get an instance of the shared viewModel
    Make the activity the store owner of the viewModel
    to ensure that the same viewModel instance is used for all screens */
 val userViewModel = viewModel<UserViewModel>(viewModelStoreOwner =
 LocalContext.current as ComponentActivity)
 val user = userViewModel.getUser(userId)
... }
```

ViewModel scoped to the Activity

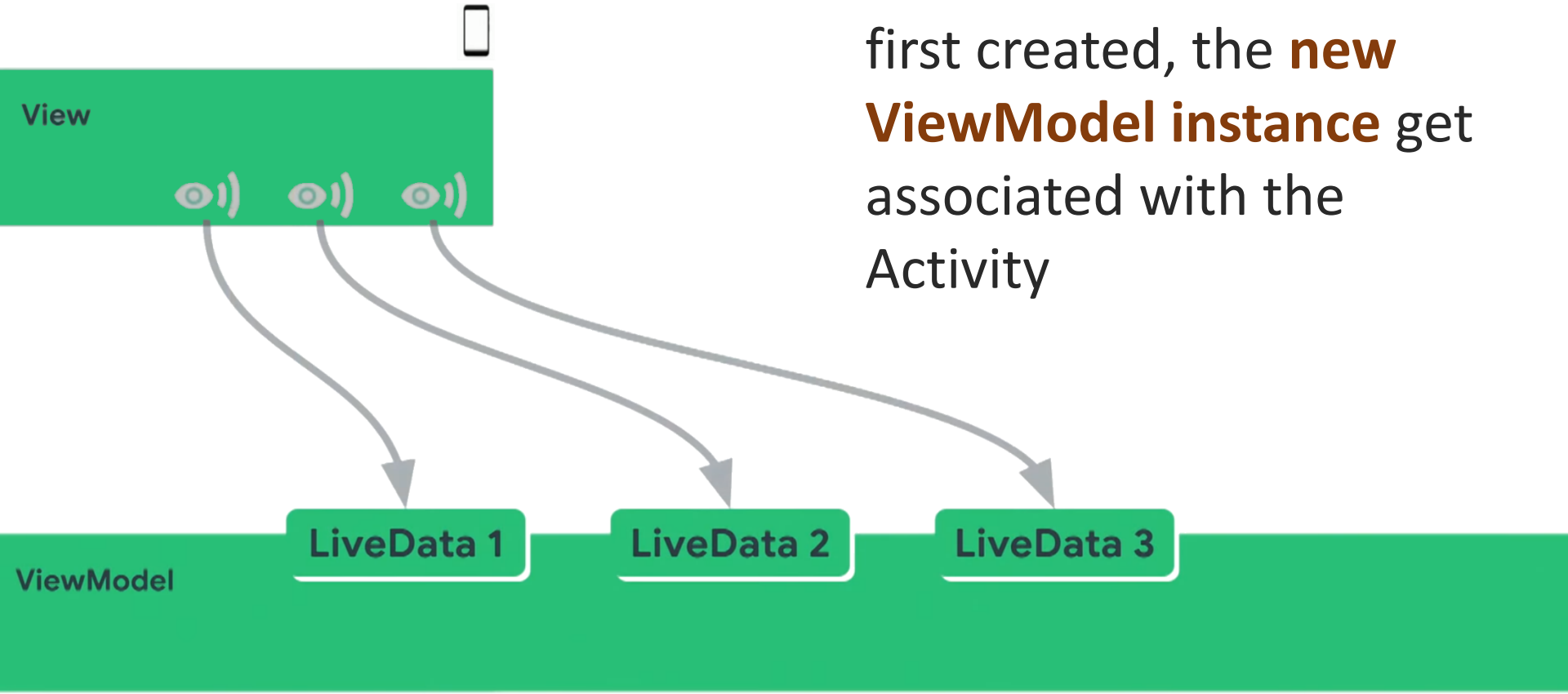
- Use **`viewModel()`** method to get an instance of the ViewModel

```
val scoreViewModel = viewModel<ScoreViewModel>
        (viewModelStoreOwner = LocalContext.current as ComponentActivity)
```

  - For the first call, this creates and returns a new ViewModel instance **scoped** to the activity (i.e., associated with the Main Activity) to make it **shared** and accessible from all screens

  - For subsequent calls, `viewModel()` method will return the pre-existing ViewModel scoped to the Activity

When the ViewModel is first Created

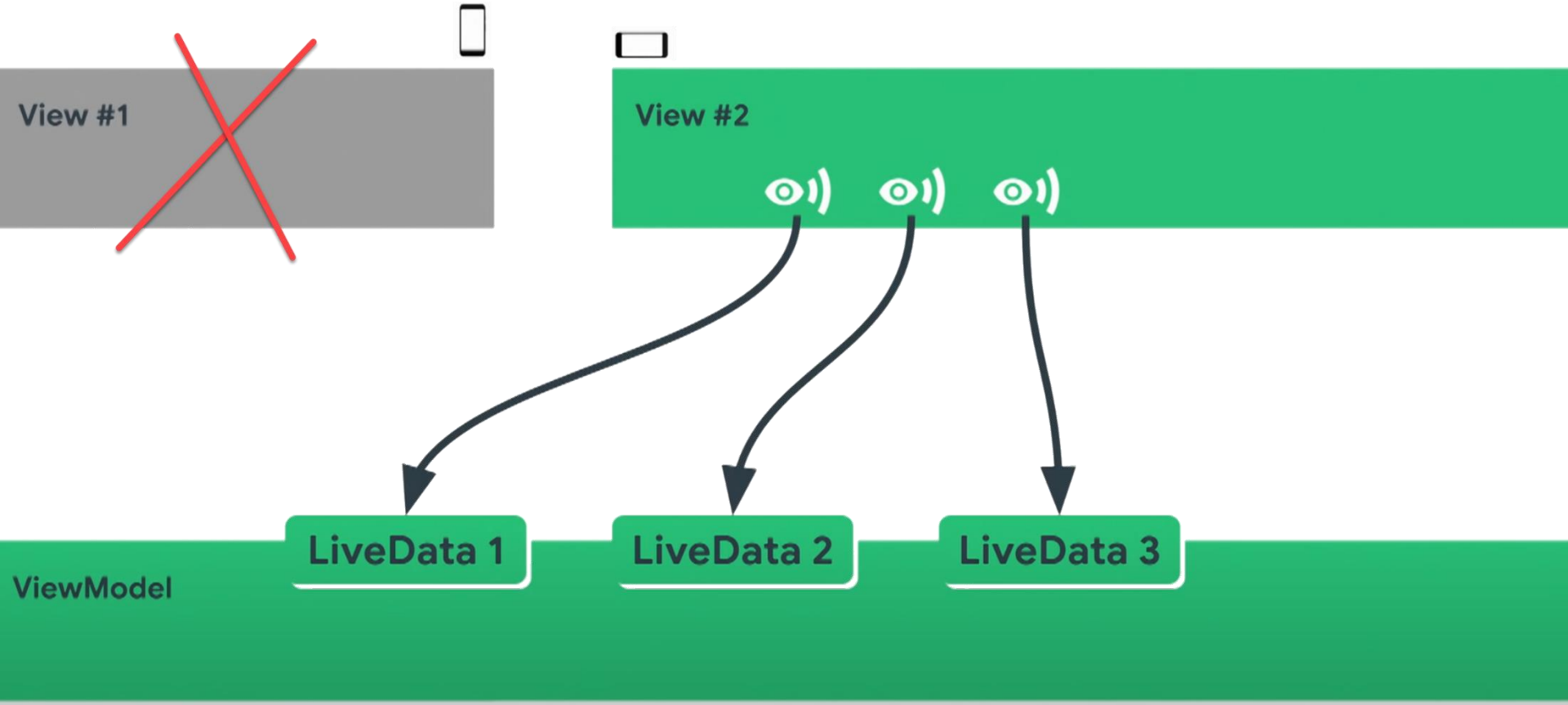


When the ViewModel is first created, the **new ViewModel instance** get associated with the Activity

Source: https://www.youtube.com/watch?v=B8ppnjGPAGE

OnConfig change (e.g., Screen Rotates)

OnConfig change, the View is destroyed, and a new instance of the View is created then it obtains **the same ViewModel instance used previously**



Source: https://www.youtube.com/watch?v=B8ppnjGPAGE

“no contexts in ViewModels” rule

- ViewModel should **not be aware of the View** who is interacting with
  => It should be decoupled from the View

  - ViewModel should not hold a reference to Activities or Views (i.e. Composables)
  - Should not have any Android framework related code
  - As this defeats the purpose of separating the UI from the data
  - Can lead to memory leaks and crashes (due to null pointer exceptions) as the ViewModel outlives the View
    - if you rotate an Activity 3 times, 3 three different Activity instances will be created, but you only have one ViewModel instance

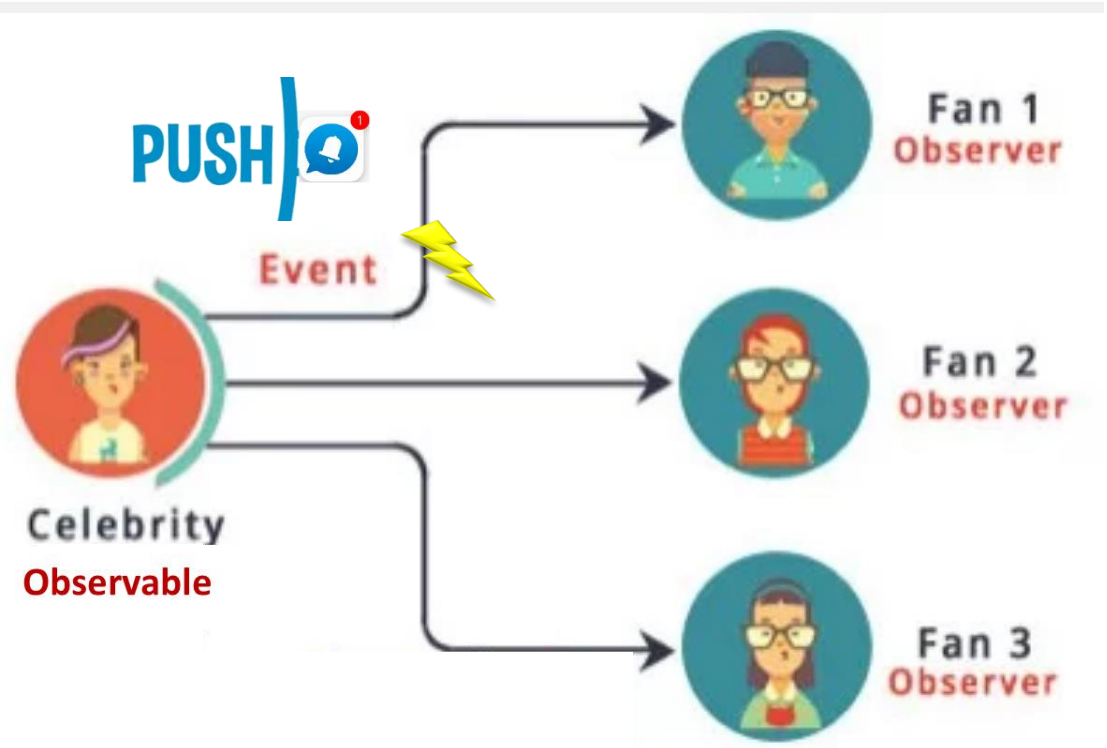
State variables

State variables

- State in an app is any value that can change over time
- A **State** variable is an **observable data holder** whose reads and writes are observed by Compose to trigger UI recomposition
  - State variable warps around an object and allows the view to **observe** it
- The ViewModel exposes **State** variables that the View observes and update the UI accordingly
  - This decouples the ViewModel from the View: the ViewModel does NOT have any direct reference to the View
  - The View can observe the ViewModel State variables for changes then update the UI (aka recomposition)

Observable - Real-Life Example

- A celebrity who has many fans on Instagram. Fans want to get all the latest updates (photos, videos, posts etc.). Here fans are **Observers** and celebrity is an **Observable**



Example - State variable

```
class ScoreViewModel : ViewModel() {
    // Private mutable state variables
    private var _team1Score = mutableStateOf(0)
    // Public State variables
    val team1Score : State<Int> = _team1Score
    fun onIncrementTeam1Score() { _team1Score.value++  }
```

```
@Composable
fun ScoreScreen() {
    // Get an instance of the ScoreViewModel
     val scoreViewModel = viewModel<ScoreViewModel>()
    Text(text = scoreViewModel.team1Score.value)
    Button(onClick = { scoreViewModel.onIncrementTeam1Score() }) {
       Text(text = "+1")
    }
    ...
}
```

Other Observable Types

- `mutableStateListOf()` creates an instance of MutableList that is observable
- `mutableStateMapOf()` creates an instance of MutableMap<K, V> that is observable
- `LiveData`: is lifecycle-aware observable data holder class
  - Meaning that LiveData only it sends updates to app component observers that are in an active lifecycle state
- `Flow`: a flow is a type that can emit a stream of values overtime
  - e.g., you can use a flow to receive live updates from a database

LiveData


View
ViewModel
State
Repository
LiveData
Data source
LiveData

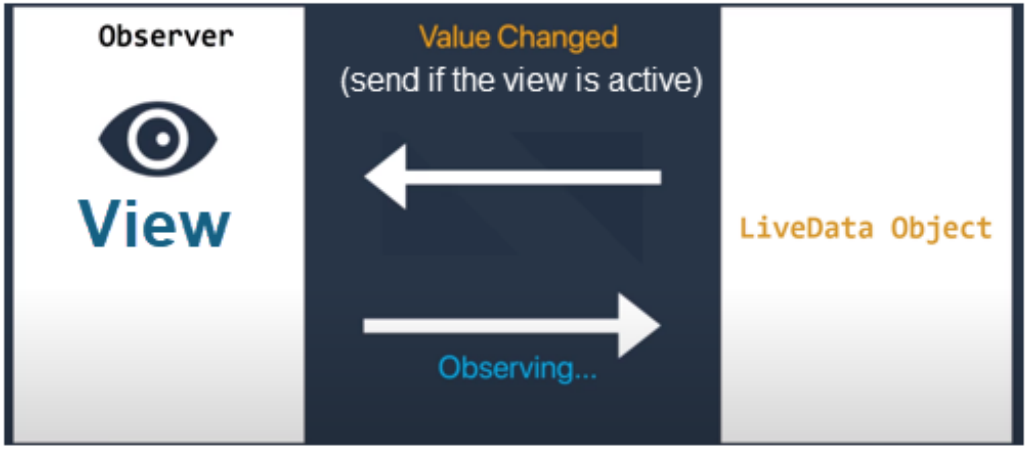
LiveData

- **`LiveData`** is an **observable data holder** (typically returned by the Repository): **active** observers (i.e., the ViewModel) get notified when data change
- The Repository return **LiveData** objects that the ViewModel can observe and notify the UI accordingly
  - This decouples the Repository from the ViewModel: The Repository does NOT have any direct reference to the ViewModel

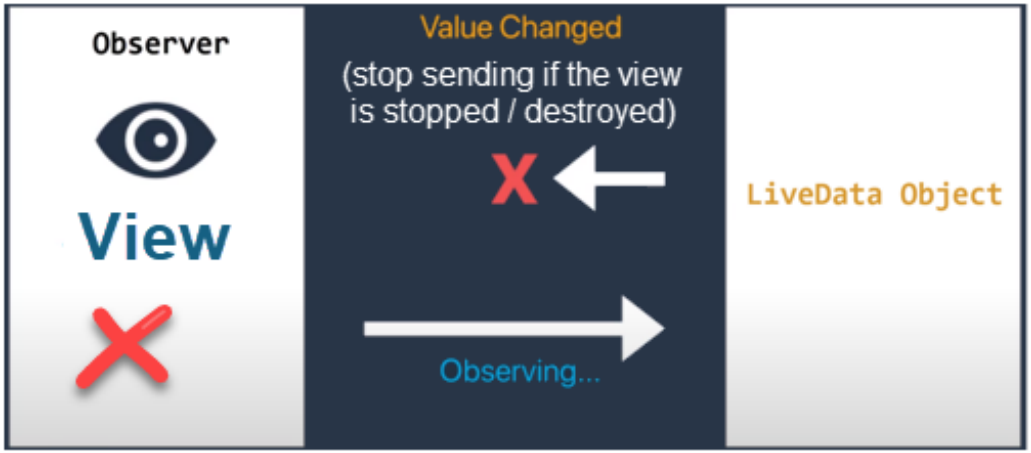
LiveData is lifecycle-aware

LiveData is aware of the Lifecycle of its Observer

- Notifies data changes to only **active** observers (Stopped/Destroyed View will NOT receive updates)

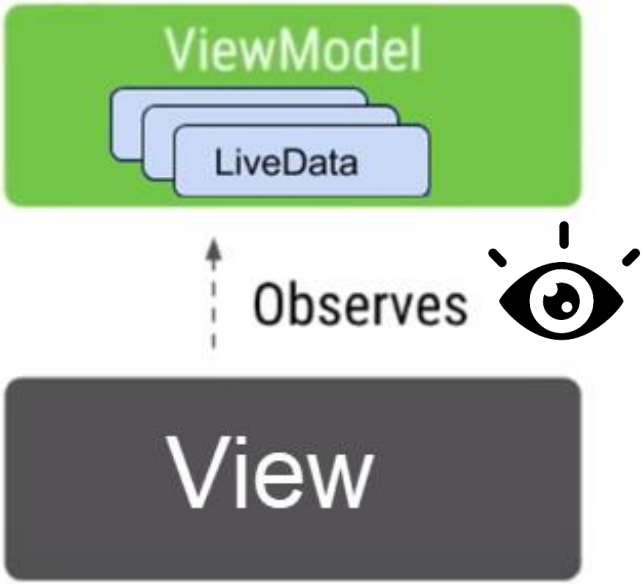


- It automatically removes the subscription when the observer is destroyed so it will not get any updates



LiveData in Code

LiveData **warps around** an object and allows the view to **observe** it



- ViewModel expose LiveData objects that the View can observe

```
object DataRepository {
    // In a real app the LiveData will be returned by
    // a database or Web API as we will see later
    fun getRedCardsCount(): LiveData<Int> = liveData {
        emit(2)
    }
}
```

```
class ScoreViewModel : ViewModel() {
    val redCardsCount = DataRepository.getRedCardsCount()
}
```

- View **observes** LiveData changes

```
@Composable
fun ScoreScreen() {
    val scoreViewModel = viewModel<ScoreViewModel>()
    val redCardsCount = scoreViewModel.redCardsCount
                                .observeAsState(0)
// Recomposes whenever redCardsCount changes
Text(
  text = "Red cards count: ${redCardsCount.value}")
}  ... }
```

LiveData.observeAsState()

- LiveData.`observeAsState()` registers as a **listener** and represent the values as a State
  - Whenever a new value is emitted, Compose recomposes those parts of the UI where that state.value is used
  - Required dependency in `build.gradle`:

```
implementation "androidx.compose.runtime:runtime-livedata:$compose_version"
```

Flow

What is Flow?

- Stream of values (produced one by one over time instead of all at once)
  - as opposed to functions that return only a single value
  - Values could be generated from network requests or database calls
- Can transform a flow using operators like map, filter, etc.

```
fun stream(): Flow<String> = flow {
    emit("🐊") // Emits the value upstream ☝
    emit("🎡")
    emit("🎉")
}
```


```
object WeatherRepository {
    private val weatherConditions = listOf("Sunny", "Windy", "Rainy", "Snowy")
    fun getWeather(): Flow<String> =
        flow {
            var counter = 0
            while (true) {
                counter++
                delay(3000)
                emit(weatherConditions[counter % weatherConditions.size])
            }
        }
}
```

```
class WeatherViewModel : ViewModel() {
    val weatherFlow: Flow<String> = WeatherRepository.getWeather()
}
```

```
@Composable
fun ScoreScreen() {
    val weatherViewModel = viewModel<WeatherViewModel>()
    val weatherFlow = weatherViewModel.weatherFlow.collectAsState(initial = "")
    // Recomposes whenever redCardsCount changes
    Text(
      text = "Red cards count: ${weatherFlow.value}" )
    }  ... }
```

Flow Operators

- Flow has operators similar to collections such as map, filter and reduce

```
(1..5).asFlow()
    .filter { it % 2 == 0 }
    .map { it * it }
    .collect { println(it.toString()) }

val result =(1..5).asFlow()
                  .reduce { a, b -> a + b }
println("result: $result")
```

Resources

- State and Jetpack Compose
  - https://developer.android.com/jetpack/compose/state
- Kotlin flows on Android
  - https://developer.android.com/kotlin/flow
- MVVM
  - https://developer.android.com/jetpack/guide
  - https://medium.com/androiddevelopers/viewmodels-a-simple-example-ed5ac416317e